Chapitre II

Introduction aux bases de données relationnelles parallèles

1 Objectifs des bases de données relationnelles parallèles

Ces dernières années, de nombreux efforts ont été entrepris par des centres de recherche universitaires et des industriels, dans le domaine de conception et de réalisation des systèmes de gestion de bases de données SGBD relationnels parallèles. Le principal objectif des SGBD parallèles est d'exploiter le parallélisme et les récentes architectures parallèles [COS 93, GER 91, JAC 93] afin d'obtenir de hautes performances [VAL 93]. Celles-ci consistent à [DEW 92] :

1- *assurer le meilleur rapport coût/performance* par rapport à la solution gros systèmes *" Mainframe "* tels que DPS8/GCOS, IBM3039/VMS, ... ,

2- *optimiser le temps de réponse des requêtes* en exploitant efficacement les niveaux du parallélisme (intra-requête, inter-requête), les formes du parallélisme (intra-opération, inter-opération) et les approches de partitionnement de données (partitionnement total, partitionnement partiel, cf II.3). Ainsi la bande passante en traitement (CPU) et la bande passante en entrées/sorties (E/S) se trouvent améliorées. Un résultat important obtenu concerne l'ordre d'influence des niveaux et des formes de parallélisme sur l'amélioration du temps de réponse. En effet, le parallélisme intra-opération a la plus grande influence sur la réduction du temps de réponse (figure II.1) ;

3- *augmenter les capacités du système parallèle* en assurant une gestion efficace des ressources. Cette augmentation est le plus fortement influencée par le parallélisme inter-requête et le moins par le parallélisme intra-opération (figure II.1) ;

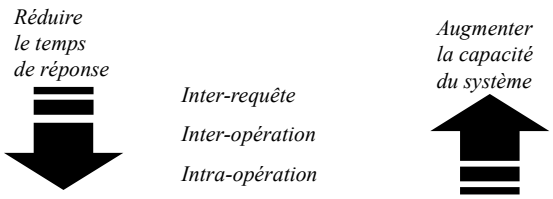


Figure II.1 : Impact des formes de parallélisme sur le temps de réponse et sur la capacité du système

4- *maintenir un bon dimensionnement évolutif " Scalability " du système parallèle*, ce qui consiste à préserver les mêmes performances du système parallèle lorsque l'on ajoute de nouvelles ressources et de nouveaux utilisateurs. D'après [DEW 92], un système parallèle idéal doit posséder la propriété de dimensionnement évolutif linéaire *" Linear scaleup "*, que l'on peut définir comme l'aptitude qu'a un système N fois plus gros (ajout de ressources CPU, disques et mémoires) à réaliser un " travail " N fois plus volumineux en une durée égale à celle obtenue par le système original. Lorsque ce " travail " est une requête, on parlera de dimensionnement évolutif batch, ce qui revient à étudier la soumission de la même requête à une base de données N fois plus volumineuse. Par contre, si ce " travail " consiste à satisfaire plusieurs petites requêtes indépendantes lancées par plusieurs clients sur une base de données partagée, on considérera le dimensionnement évolutif transactionnel qui correspond à étudier la satisfaction de N fois plus de requêtes formulées par N fois plus de clients destinées à une base de données N fois plus volumineuse.

L'objectif de ce chapitre est d'introduire les différentes notions de base qui facilitent la compréhension des chapitres suivants. Son organisation est articulée comme suit : nous présentons d'abord les différents modèles d'architecture parallèle, en particulier les architectures multiprocesseurs à mémoire commune, à disques partagés et à mémoire distribuée. Ensuite, nous décrivons les approches de partitionnement d'un ensemble de relations dans un SGBD parallèle. Enfin, nous étudions les principaux algorithmes de jointure séquentiels et parallèles et présentons brièvement leur comportement, notamment par rapport à la taille des opérandes, la taille mémoire, et l'exploitation d' index.

2 Modèles d'architecture parallèle

La classification des architectures parallèles la plus reconnue actuellement est due à M.J. Flynn [FLY 72]. Celle-ci est basée sur la façon dont les flots de contrôle (instruction stream) et de données sont réparties sur les processeurs [JAC 93, GER 91]. Deux classes d'architectures parallèles ont été conçues et réalisées par les constructeurs. Les architectures parallèles fondées sur le modèle d'exécution SIMD (Single Instruction Multiple Data Stream) et les architectures parallèles fondées sur le modèle d'exécution MIMD (Multiple Instruction Multiple Data Stream).

Les architectures qui fonctionnent en mode SIMD possèdent une seule unité de commande, mais les unités de traitement sont dupliquées. Dans ce modèle d'exécution toutes les unités de traitement effectuent la même opération au même moment sur des données différentes. Il y a donc une exécution parallèle synchrone des opérations sous le contrôle d'une unité de contrôle. Parmi les principaux produits les plus connus nous citons [JAC 93] : La Connection Machine commercialisée par Thinking Machine Corp. [HIL 85], GAPP (Geometrical and Arithmetical Parallel Processor) commercialisée par NCR, DAP (Distributed Array Processor) et MPP (Massively Parallel Processor) construite par GOODYEAR AEROSPACE.

Quant aux architectures de type MIMD, chaque processeur (i.e. unité de commande et unité de traitement) est capable d'exécuter des opérations différentes de celles qu'exécutent les autres processeurs. De plus, en fonction de l'organisation des données en mémoire, ces architectures peuvent être *fortement couplées*, dans le cas où les processeurs peuvent accéder à une mémoire commune, ou *faiblement couplées*, dans le cas où à chaque processeur est associé une mémoire locale. On obtient donc des systèmes multiprocesseurs à mémoire commune (couplage fort) et des systèmes multiprocesseurs à mémoire distribuée (couplage faible). Les machines Butterfly de BBN, de Sequent Computer Systems (Balance et Symetry) et de Alliant Computer Systems font parties de la famille des multiprocesseurs à mémoire partagée. L'IPSC/2, CAP-C5, SUPERNODE (T800) sont font parties de la famille des multiprocesseurs à mémoire distribuée.

Quelle que soit la classe d'architecture multiprocesseur les organes (i.e. processeur, mémoire,...) contenant les programmes et les données sont amenés à communiquer des données entre eux. Le temps nécessaire pour faire communiquer deux processeurs ou un processeur et une mémoire influe fortement sur les performances des algorithmes parallèles. Il devient donc nécessaire de définir une structure de communication entre les organes de la machine parallèle, appelée *topologie du réseau d'interconnexion* [FLE 95]. Plusieurs topo-

logies d'interconnexion ont été déjà conçues, réalisées et exploitées dont l'idée directrice est toujours de trouver le meilleur compromis entre *l'efficacité des communications et le nombre de liaisons entre processeurs*. On distingue deux classes de réseaux d'interconnexion : celle à topologie fixe et celle à topologie reconfigurable permettant d'une part, une adaptation à une large classe d'applications et d'autre part, de modifier dynamiquement la configuration du réseau d'interconnexion. Un réseau d'interconnexion peut être caractérisé par 3 éléments [JAC 93] : la *distance* entre deux processeurs (i.e le plus court chemin pour aller d'un processeur à un autre) le *diamètre* (i.e le plus long chemin existant entre deux processeurs) et la *connectivité* (i.e le nombre de chemins indépendants joignant deux processeurs).

Dans un contexte de bases de données les applications manipulent de plus en plus de gros volumes de données devant être stockées sur *des mémoires secondaires (disques)*. Dans les systèmes relationnels monoprocesseur le problème de l'optimisation des Entrées/Sorties disque a un impact considérable sur les performances en terme de temps de réponse des requêtes. En ce qui concerne les systèmes relationnels parallèles, le problème est encore plus important et plus complexe. En effet, le fait que les disques puissent être partagés entre plusieurs processeurs au travers d'un réseau d'interconnexion, pose le problème du *goulot d'étranglement* que forment les entrées /sorties. Par conséquent, la façon dont les processeurs, la mémoire et les disques sont reliés affecte fortement les performances des algorithmes de traitement parallèles des requêtes.

La classification des architectures parallèles, due à Flynn, a permis de donner une définition d'un SGBD parallèle. Celui-ci est un SGBD classique implanté sur une architecture parallèle fondée sur un modèle d'exécution de type MIMD. Trois grandes familles de SGBD parallèles ont été proposées : les SGBD parallèles basés sur une architecture multiprocesseur à mémoire commune (figure II.2), les SGBD parallèles fondés sur une architecture multiprocesseur à disques partagés (figure II.3) et les SGBD parallèles s'appuyant sur une architecture multiprocesseur à mémoire distribuée (figure II.4). Parmi les principaux produits et prototypes de SGBD parallèles à mémoire distribuée nous citons [DEW 92] : DBC 1012 (Teradata) [CAR 92, WIT 93], NonStop SQL [ENG 95], DB2 Parallel Edition [BAR 95], Gamma [DEW 90], Bubba [BOR 90], EDS (European Declarative System) [CHA 92, ZIA 92], PRISMA/DB [APE 92] et le " Parallel Query " d'ORACLE 7. Dans les SGBD parallèles à disques partagés nous citons : IMS/VS Data Sharing d'IBM et ORACLE7 Parallel Server sur un cluster UNIX de noeuds[1] faiblement couplé. Un cluster est constitué de plusieurs noeuds reliés par des liaisons à haut débit et disposant d'un accès concurrent à disques partagés. Les

systèmes communiquent au travers du réseau d'interconnexion et les disques partagés. Quant aux SGBD parallèles à mémoire commune nous citons XPRS (eXtended Postgres on Raid and Sprite) [STO 88], DBS3 [BER 91], Informix-OnLine Dynamic Server avec Parallel Data Query (basé sur l'architecture Dynamic Scalable Architecture) [INF 95] et le système Volcano [GRA 90].

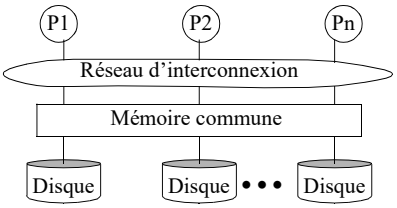


Figure II.2 : Architecture multiprocesseur à mémoire commune

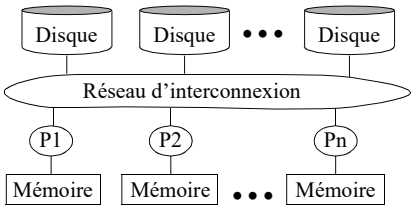


Figure II.3 : Architecture multiprocesseur à disques partagés

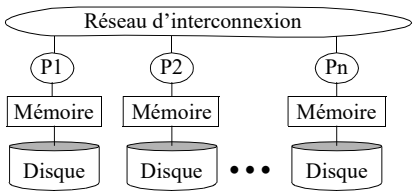


Figure II.4 : Architecture multiprocesseur à mémoire distribuée

Les trois types d'architectures parallèles présentent des avantages et des inconvénients [DEW 92, VAL 93]. En effet, les architectures à mémoire distribuée réduisent les interférences par la minimisation des ressources parta-

1. Un noeud est constitué d'une instance ORACLE Parallel Server, d'un Distributed Lock Manager, d'un cache et d'une application.

gées. Les processeurs et les mémoires sont exploités indépendamment du réseau d'interconnexion. De plus, les architectures à mémoire distribuée sont plus facilement *extensibles*, et plus *fiable*s par rapport aux architectures à mémoire commune lesquelles présentent l'avantage d'un faible coût de communication et leur relative facilité de programmation. En effet la plupart des techniques développées dans un contexte multitâches peuvent être exploitées [JAC 93]. Dans une architecture à mémoire commune, le réseau d'interconnexion est accédé systématiquement pour tout accès mémoire par un processeur. Ainsi, dans une telle architecture, le problème d'interférence devient critique (puisque le réseau d'interconnexion doit avoir une bande passante correspondant à la somme des processeurs) et le nombre de processeurs se trouve ainsi limité. Les principaux avantages des architectures à disques partagés par rapport aux architectures à mémoire commune sont :

- une meilleure extensibilité dans le sens où chaque processeur possède une mémoire privée et, aucune réécriture des programmes d'applications n'est nécessaire puisqu'ils continuent à accéder à la même base de données,
- une meilleure *disponibilité* : en cas de problème sur un des noeuds la base de données reste disponible. Par exemple dans le système ORACLE7 Parallel Server en cas de défaillance d'un noeud un processus appelé SMON (System Monitor) d'un autre noeud détecte le problème et réalise une *récupération automatique* du noeud concerné. Durant cette opération les autres noeuds et la base de données continuent leurs traitements.

Récemment une autre famille d'architecture parallèle, appelée architecture hybride (figure II.5), a été initialement proposée par Richardson et al. [RIC 87] et ses performances ont été analysées au travers un modèle analytique proposé par [HUA 91]. Une architecture hybride consiste, par exemple, en un système de clusters (i.e. grappes) faiblement couplé dans lequel un cluster est un système multiprocesseur fortement couplé. Cette architecture semble prometteuse et particulièrement efficace [VAL 93] pour réaliser des SGBD parallèles dans la mesure où elle tente de combiner les avantages des architectures à mémoire commune avec la puissance, due à l'extensibilité, des architectures à mémoire distribuée.

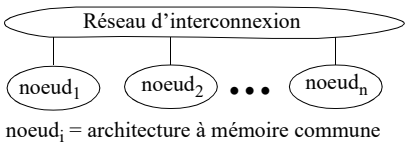


Figure II.5 : Architecture hybride

3 Stratégies de répartition des données

3. 1 Motivation

La répartition d'une relation consiste à distribuer les tuples de la relation sur plusieurs disques [DEW 92]. Dans un système monoprocesseur, la répartition d'une relation était nécessaire soit parce que le fichier était trop important pour tenir sur un seul disque, soit parce que le nombre d'accès à un fichier ne pouvait pas être supporté par un seul disque.

Dans un système parallèle, la répartition des relations sur plusieurs disques permet :

- d'augmenter la bande passante en E/S en exploitant au maximum le parallélisme des opérations de lecture/écriture d'une ou plusieurs relations,
- d'effectuer des traitements parallèles le plus proche possible des données,
- de favoriser l'équilibrage de charge pour maximiser l'utilisation des ressources du système parallèle.

Le principal problème qui se pose au niveau de la répartition des données, appelé aussi *placement de données*, consiste à déterminer et à maintenir le meilleur compromis entre les traitements et les communications [COP 88]. Il existe deux approches pour résoudre le problème de placement d'un ensemble de relations dans un système parallèle.

3. 2 Approches et méthodes de répartition

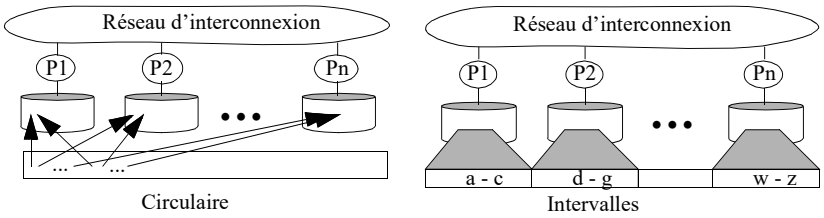
La première approche, appelée *partitionnement total* " Full desclutering "[LIV 87], consiste à répartir horizontalement chaque relation de base sur tous les disques du système. Elle est adoptée dans les SGBD parallèles à mémoire commune, par exemple le système XPRS [HON 92]. Quant aux SGBD parallèles à mémoire distribuée, cette approche peut être exploitée si le nombre de processeurs est relativement petit (~ <32), étant donné que l'on ne risque pas de rencontrer le problème relatif au compromis *traitement-communication*. En effet, à partir d'un certain nombre de processeurs et en fonction de la taille des relations, le temps de traitement gagné par l'ajout d'un processeur est inférieur au temps perdu par les communications. Malheureusement, les performances de cette première approche sont limitées dans le cas des opérations complexes comme les jointures. En effet, on ne peut pas exécuter en parallèle une opération de jointure et une opération de répartition puisque cette dernière s'effectue sur l'ensemble des processeurs du système.

La seconde approche, appelée *partitionnement partiel* " Partial desclutering ", consiste à répartir chaque relation de base sur un *sous-*

ensemble de disques. Cette approche est principalement adoptée dans les SGBD parallèles à mémoire distribuée qui comportent un nombre important de noeuds[1], comme dans le système parallèle Bubba [BOR 90]. Le nombre de noeuds pour répartir une relation est déterminé en fonction de la taille de la relation [HAM 92b] et éventuellement de la fréquence d'accès par les applications [COP 88]. Ce type de partitionnement offre de meilleures performances dans la mesure où il est possible d'exécuter en parallèle une opération de répartition d'une relation, et une opération quelconque (notamment une autre opération de répartition) puisqu'une opération de répartition n'occupe plus la totalité des noeuds du système. En revanche, cette approche est plus difficile à réaliser et à administrer en raison de la difficulté d'estimer correctement le *degré de répartition des relations* (et sa maintenance) qui minimise le temps de réponse et qui maximise l'utilisation des ressources du système parallèle [COP 88, NIC 91].

Cependant, quelle que soit l'approche de placement de données préconisée, il existe plusieurs méthodes de répartition de données. Généralement, dans les systèmes parallèles trois méthodes (figure II.6) sont proposées à l'administrateur pour répartir les données horizontalement [LIV 87, NIC 91, DEW 92] :

- répartition circulaire " Round Robin " : les données sont réparties au fur et à mesure, soit tuple à tuple soit page par page, sur les disques et, a priori, sans moyen direct de les retrouver ultérieurement,
- utilisation d'une fonction de hachage " Hashing " qui à un ou plusieurs attributs d'une relation (généralement une clé pour réduire le problème d'une mauvaise répartition " Data skew ") associe un numéro de disque. Par exemple, la fonction modulo *x mod v* (i.e. le reste de la division entière de x par v) est souvent utilisée ;
- utilisation d'intervalles " Range Partitioning " spécifiés par l'administrateur au moyen de prédicats de fragmentation (i.e. opérations de restriction).



1. Un noeud est constitué d'un processeur, d'une RAM et d'un ou plusieurs disques.

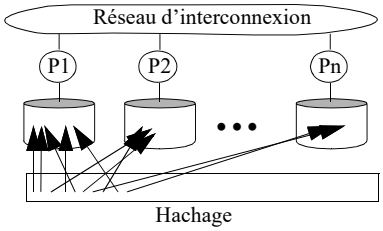


Figure II.6 : Méthodes de répartition des données [DEW 92]

Les méthodes de répartition des données, répartition circulaire, par hachage et par intervalles sont respectivement exploitées efficacement dans le cas où les applications [DEW 92] :

- utilisent beaucoup d'accès séquentiels aux relations,
- nécessitent des accès séquentiels et directs sur les attributs de hachage,
- exploitent uniformément les accès séquentiels, directs et le regroupement physique de données " Clustering ".

Ces méthodes présentent quelques inconvénients qui ont un impact considérable sur l'équilibre de charge. En effet, pour le méthode de répartition par hachage implantée dans les systèmes parallèles comme Bubba, Gamma, et la DBC1012, la qualité de répartition dépend du choix d'une bonne fonction de hachage devant fournir une répartition bien homogène. En ce qui concerne la méthode des intervalles implantée dans les systèmes parallèles Bubba, Gamma, et NonStop SQL, deux critiques peuvent être exprimées [NIC 91] :

- la construction d'une répartition relativement bien équilibrée dépend fortement des informations quantitatives que possède l'administrateur telles que : le profil de la base de données (cardinalité des relations, longueur d'un tuple,...), les caractéristiques de requêtes (type d'utilisation, taille d'une requête et la forme d'une requête), et la topologie du réseau d'interconnexion,
- la possibilité de choisir les prédicats de fragmentation risque de poser le problème d'une *mauvaise répartition* " Data skew " (i. e. distribution non-uniforme des valeurs de l'attribut de répartition) en regroupant des tuples qui peuvent être traités en parallèle.

Pour atténuer les conséquences dues à une mauvaise répartition des données, Ghandeharizadeh et DeWitt [GHA 90] ont proposé une méthode de *répartition hybride*. Celle-ci est fondée sur :

- la définition d'une requête moyenne Q_{moy} représentative de toutes les requêtes de sélection sur la relation R en tenant compte de leurs fréquences,
- la détermination du nombre de tuples par fragment TF qui dépend du nombre de tuples résultats $TupQ_{moy}$ et du nombre de processeurs d pour lequel le temps de réponse pour Q_{moy} est minimal.

Pour répartir la relation R, le nombre de tuples $TF = TupQ_{moy}/d$ par fragment est tout d'abord calculé. Ensuite, la relation R est triée, puis des fragments contenant approximativement TF tuples sont construits avec une fonction de répartition de type intervalle. Enfin, les fragments sont distribués circulairement " Round Robin " sur les disques du système parallèle de manière à assurer que d fragments adjacents soient stockés sur des disques différents.

Avec cette méthode, le traitement d'une requête de sélection s'effectue toujours au voisinage du nombre de processeurs optimal d. Cependant, si la clause sélective ne porte pas sur l'attribut de répartition, alors la requête doit s'effectuer sur tous les processeurs.

Une exploitation efficace des méthodes de répartition implique une gestion rigoureuse qui tient compte principalement de deux éléments :

1- *extensibilité du dictionnaire de données* dans le but d'intégrer de nouvelles informations telles que les caractéristiques de requêtes, les paramètres du modèle d'architecture parallèle et du système, ainsi que les modèles de coûts. Ces informations permettent d'évaluer des métriques nécessaires pour la prise de décisions par le SGBD ou par l'administrateur de bases données ;

2- *migration des données* qui consiste à prendre en compte la modification de la répartition de données à cause de l'évolution des données et de l'extensibilité du système parallèle.

4 Niveaux, formes et types de parallélisme

A partir des approches de placement et des méthodes de répartition deux *niveaux de parallélisme* peuvent être définis : le parallélisme inter-requête et le parallélisme intra-requête [GAR 90, LU 94]. Le parallélisme inter-requête permet d'exécuter plusieurs requêtes en parallèle par des processeurs. En ce qui concerne le parallélisme intra-requête, plusieurs processeurs coopèrent pour exécuter en parallèle les opérations d'une même requête. Dans ce dernier niveau, deux *formes* de parallélisme peuvent être extraites : (i) le parallélisme *intra-opération* permettant l'exécution parallèle d'une même opération sur

des données obtenues par une méthode de répartition et (ii) le parallélisme *inter-opération* permettant l'exécution parallèle de plusieurs opérations d'une même requête. Dans le parallélisme inter-opération, on peut dégager deux *types de parallélisme* : le parallélisme *indépendant et* le parallélisme *dépendant* (appelé aussi parallélisme *pipeline ou de flux*). Le parallélisme dépendant correspond à une exécution parallèle des opérations qui communiquent.

Nous illustrons les définitions introduites précédemment au travers de l'expression algébrique (R1 ⋈ R2) ⋈ (R3 ⋈ R4), en s'inspirant de la définition du profil de parallélisme d'une application donnée par [GER 91]. En effet, les auteurs ont mis en évidence une correspondance entre les types de parallélisme et deux paramètres caractéristiques d'un arbre (largeur et hauteur) associé à un programme. Le parallélisme indépendant (resp. dépendant ou pipeline) est lié à la largeur (resp. hauteur) de l'arbre. La figure II.7 fournit une représentation graphique des formes et types de parallélisme, en supposant qu'il n'y a pas de contention de ressources.

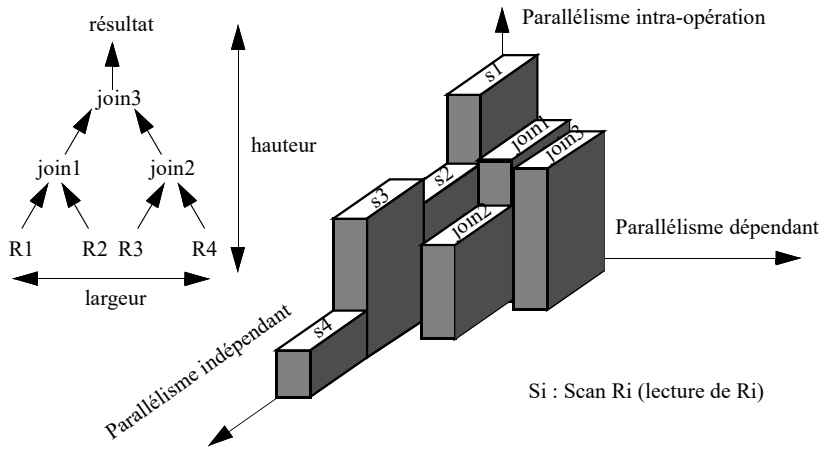


Figure II.7 : Représentation graphique des formes et des types de parallélisme

Dans la section suivante, après avoir décrit les principaux algorithmes séquentiels de jointure, nous nous intéressons à leurs versions parallèles en exploitant les méthodes de répartition de données. L'opération de jointure est l'opération relationnelle dont la fréquence d'utilisation est la plus importante et dont les performances sont déterminantes pour la viabilité de tout système de gestion de bases de données.

5 Opérateurs de jointure

Généralement il existe trois classes d'algorithmes pour calculer la jointure de deux relations R et S [KOR 88, LU 94, ULL 85]. La première classe, appelée *jointure par produit cartésien,* ou encore *jointure par boucles imbriquées,* consiste à comparer tous les tuples de la relation R avec tous les tuples de la relation S. Une deuxième classe, appelée *jointure par tri-fusion,* trie d'abord les deux relations sur leur attribut de jointure puis parcourt les deux relations en profitant de l'ordre du tri pour déterminer les tuples satisfaisant la condition de jointure. Une dernière classe, appelée *jointure par hachage* [KIT 83, BRA 84], construit une table de hachage avec la relation R et balaie la relation S de manière séquentielle en sondant la table de hachage afin de trouver les tuples pertinents pour calculer la jointure.

Dans cette section, l'étude de quelques algorithmes parallèles de jointure est basée sur une architecture à mémoire distribuée de Dmax processeurs ayant chacun au moins un disque. Les relations de base sont initialement réparties en suivant soit l'approche de partitionnement total soit l'approche de partionnement partiel. De plus, la taille en page de la mémoire locale d'un processeur est égale à b+1 pages décomposée généralement en un tampon principal de b-1 pages et 2 tampons auxiliaires. Le nombre de processeurs alloué à une opération de jointure parallèle est d avec d <= Dmax.

5. 1 Jointure par produit cartésien

Le principe de l'algorithme de jointure par produit cartésien est fondé sur la comparaison de chaque tuple de la relation R (relation externe) avec tous les tuples de la relation S (relation interne). Les tuples vérifiant la condition de jointure forment le résultat. L'algorithme général [BOU 90, DEL 82, KOR 88, ULL 85] est décrit ci-après.

```
Procédure jointure_prod_cartésien (R, S, T : relation; cond : condition_de_jointure)
Début
    pour chaque tuple r ∈ R faire
        pour chaque tuple s ∈ S faire
            si cond  alors écrire le tuple résultat dans T; fin si
        fin pour
    fin pour
Fin
```

Figure II.8 : Jointure par produit cartésien

L'inconvénient majeur de cet algorithme réside dans le nombre important d'entrées/sorties disque engendrées dans le cas où les deux relations $||R||^1$ >b-1 et ||S|| >b-1. Afin de réduire les entrées/sorties, la relation R est chargée en

mémoire par morceaux de (b-1) pages. Tous les tuples de la relation R en mémoire sont comparés à tous les tuples de la relation S, en la balayant, page par page, dans un tampon de taille 1 page. Les tuples vérifiant la condition de jointure sont écrits dans un tampon résultat de taille 1 page. Lorsque ce tampon est plein, il déclenche son écriture sur disque. On obtient une nouvelle version de l'algorithme ci-dessus, appelé *jointure par produit cartésien par bloc*.

```
Procédure jointure_prod_cartésien_bloc (R, S, T : relation; cond : condition_de_jointure)
Début
    tant que non_fin (R) faire
        lire (b-1) pages de R en mémoire;
        tant que non_fin (S) faire
            lire 1 page de S en mémoire;
                pour chaque tuple r ∈ R en mémoire faire
                    pour chaque tuple s ∈ S en mémoire faire
                        si cond  alors
                            écrire le tuple résultat dans tampon_res;
                            si tampon_res est plein alors écrire tampon_res sur disque dans T; fin si
                        fin si
                    fin pour
                fin pour
        fin tant que
    fin tant que
    écrire tampon_res sur disque dans T;
Fin
```

Figure II.9 : Jointure par produit cartésien par bloc

Le principal problème des algorithmes ci-dessus (figure II.8 et figure II.9) est que pour chaque tuple de R la totalité de la relation S est balayée. Ce problème peut être résolu par l'utilisation d'un index sur l'attribut de jointure de la relation interne de la forme (attribut, pointeur). L'index est en fait utilisé pour rechercher efficacement les tuples de S vérifiant la condition de jointure r.a = s.b, puisque seule les pages pertinentes de S sont lues en mémoire [KOR 88]. Cette technique permet de réduire considérablement le nombre d'entrées/sorties et le nombre de comparaisons.

L'algorithme de jointure par produit cartésien indexé (figure II.10), peut encore être amélioré de deux manières en exploitant soit un *cluster à index* [ABD 95] qui regroupe tous les tuples vérifiant r.a = s.b soit un cluster à index et le tri de la relation externe R. Cette dernière optimisation assure une lecture des pages du cluster à index dans l'ordre, ce qui permet d'éviter de *relire* ces pages.

1. ||R|| = nombre de pages de R.

```
Procédure jointure_prod_cartésien_index (R, S, T : relation; table_index : index)
Début
   tant que non_fin (R) faire
      lire 1 page de R;
      pour chaque tuple r ∈ R en mémoire faire
         chercher_entree_index (r.a, table_index, ptidx);
         /* ptidx : pointeur sur la table d'index */
         tant que non_fin (table_index) et ptidx.b = r.a faire
            charger la page correpondant à ptidx.ref;
            /* ptidx.ref : pointeur sur un tuple de S */
            écrire le tuple résultat dans tampon_res;
            si tampon_res est plein alors écrire tampon_res sur disque dans T; fin si
            suivant (ptidx);
         fin tant que
      fin pour
   fin tant que
   écrire tampon_res sur disque dans T;
Fin
```

Figure II.10 : Jointure par produit cartésien indexé

Pour paralléliser l'algorithme de jointure par produit cartésien par bloc, la relation R est redistribuée (si nécessaire) sur les disques des d processeurs avec une fonction de *répartition quelconque* en d sous-relations R_1, R_2, ..., R_d. La relation S est dupliquée sur les disques des d processeurs. Sur chaque processeur est exécuté une jointure par produit cartésien par bloc d'une sous-relation R_i et de la relation S.

Dans le but d'éviter la comparaison de toute la relation S avec chaque sous-relation R_i, la relation S est redistribuée sur les disques des d processeurs. Cette redistribution doit s'effectuer sur les relations R et S avec une *même fonction de répartition* de type hachage ou intervalle appliquée sur l'attribut de jointure. Ensuite, sur chaque processeur est exécuté l'algorithme de jointure par produit cartésien par bloc de deux sous-relation R_i et S_i. L'utilisation de la même fonction de répartition évite de dupliquer la relation S.

L'algorithme de jointure par produit cartésien indexé peut être parallélisé si la relation S est initialement répartie sur les disques des d processeurs en sous relations S_1, S_2, ..., S_d et s'il existe un index sur l'attribut de jointure sur chaque sous relation. Dans ce cas, l'algorithme parallèle consiste d'abord à dupliquer la relation R sur tous les disques des d processeurs. Ensuite, sur chaque processeur est exécuté l'algorithme de jointure par produit cartésien indexé.

Cet algorithme souffre de la duplication de la relation R. Dans le cas où la relation S est répartie sur l'attribut de jointure avec une fonction de type

hachage ou intervalles, alors il est possible d'éviter cette duplication en répartissant R avec la même fonction. L'algorithme de jointure par produit cartésien indexé est exécuté sur chaque processeur.

L'algorithme développé ci-dessus exige que la relation S soit répartie sur l'attribut de jointure avec une méthode de répartition de type hachage ou de type intervalle. Dewitt, Naughton et Burger [DEW 93] proposent un algorithme parallèle, appelé jointure par produit cartésien parallèle par répartition, *sans contrainte sur la méthode de répartition* de la relation S. L'idée de base est d'introduire une relation MAPS(#p, b) où #p est l'attribut désignant le numéro de processeur destination et b est l'attribut de jointure de S. Cette relation permet de *redistribuer* les tuples de la relation R sur les disques des processeurs P_0, P_1,..., P_{d-1} sur lesquels la relation S est répartie initialement.

La première étape de l'algorithme consiste, sur chaque processeur P_i, à insérer un tuple (i, s.b) dans la sous-relation $MAPS_i$ pour chaque tuple s de la sous-relation S_i. Dans la deuxième étape, la relation MAPS est redistribuée sur les processeurs P_0, P_1, ..., P_{d-1} en utilisant une fonction de répartition de type hachage ou de type intervalle appliquée sur l'attribut b. La troisième étape redistribue la relation R sur les processeurs P_0, P_1, ..., P_{d-1} avec la même fonction de répartition appliquée sur l'attribut de jointure de la relation R. Dans une quatrième étape, chaque processeur P_i envoie chaque tuple de la sous-relation R_i à l'ensemble des processeur m.#p tel que m.b = r.a et $m \in MAPS_i$. Enfin, dans une dernière étape, chaque processeur P_i exécute l'algorithme de jointure par produit cartésien indexé. Le principe de l'algorithme est illustré dans l'exemple schématisé par la figure II.11.

R	a	c
	1	2
	2	5
	3	1
	4	4
	5	3
	6	10
	7	11
	8	6
	9	7

Condition de jointure : R.a = S.b

S	b	e
	1	0
	1	1
	2	4
	3	3
	3	2
	4	7
	5	5
	6	6
	7	10
	7	8
	8	9
	9	11

E0 : répartition initial de la relation S

S_0	b	e
	1	0
	3	3
	6	6
	8	9

S_1	b	e
	1	1
	2	4
	4	7
	7	10

S_2	b	e
	3	2
	5	5
	7	8
	9	11

Chaque sous-relation S_i est stockée sur le disque du processeur P_i

E1 : construction de la relation MAPS

$MAPS_0$	#p	b
	0	1
	0	3
	0	6
	0	8

$MAPS_1$	#p	b
	1	1
	1	2
	1	4
	1	7

$MAPS_2$	#p	b
	2	3
	2	5
	2	7
	2	9

Chaque sous-relation MAP S_i est stockée sur le disque du processeur P_i

E2 : redistribution de la relation MAPS avec la fonction MAPS.b mod 3 = i

$MAPS_0$	#p	b
	0	3
	0	6
	2	3
	2	9

$MAPS_1$	#p	b
	0	1
	1	1
	1	4
	1	7
	2	7

$MAPS_2$	#p	b
	0	8
	1	2
	2	5

E3 : redistribution de la relation R avec la fonction R.a mod 3 = i

R_0	a	c
	3	1
	6	10
	9	7

R_1	a	c
	1	2
	4	4
	7	11

R_2	a	c
	2	5
	5	3
	8	6

Chaque sous-relation R_i est stockée sur le disque du processeur P_i

E4 : redistribution de la relation R en utilisant la relation MAPS

P0

R_0	a	c
	3	1
	6	10
	1	2
	8	6

S_0	b	e
	1	0
	3	3
	6	6
	8	9

P1

R_1	a	c
	1	2
	4	4
	7	11
	2	5

S_1	b	e
	1	1
	2	4
	4	7
	7	10

P2

R_2	a	c
	3	1
	9	7
	7	11
	5	3

S_2	b	e
	3	2
	5	5
	7	8
	9	11

Figure II.11 : Illustration de l'algorithme jointure par produit cartésien parallèle par répartition

5. 2 Jointure par tri-fusion

L'inconvénient majeur du principe de la jointure par produit cartésien est qu'il nécessite le balayage de toute la relation S pour chaque tuple de R [DEL 82]. Une approche pour réduire ce balayage est de trier les deux relations. Ceci constitue la base d'un nouvel algorithme, appelé *jointure par tri-fusion*, dont le principe est décrit ci-après. L'algorithme de jointure par tri-fusion est constitué de deux phases. La première phase trie les deux relations

R et S sur leurs attributs de jointure. Le processus de tri, appelé tri par fusion d'une relation R, est composé d'un *tri interne* et d'un *tri externe*. Le tri interne ordonne indépendamment chaque page [KNU 73, AHO 89]. Le tri externe (figure II.12) consiste à fusionner à chaque étape et à chaque itération b monotonies[1] en une seule. La première étape fusionne les ||R|| monotonies d'une page en $\lceil ||R||/b \rceil$ monotonies (où $\lceil\ \rceil$ dénote l'opération de partie entière supérieure). La seconde étape fusionne les $\lceil ||R||/b \rceil$ monotonies en $\lceil \lceil ||R||/b \rceil /b \rceil$ monotonies, etc. Le nombre total d'étape est $\lceil \log_b ||R|| \rceil$.

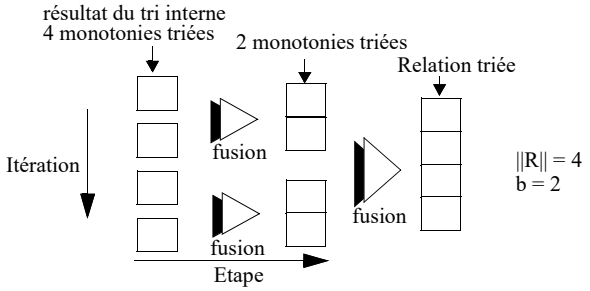


Figure II.12 : Principe du tri externe

```
Procédure jointure_tri_fusion (R, S, T : relation)
Début
   r := premier_tuple (R); s := premier_tuple (S);
   tant que (non_fin (R) et non_fin (S)) faire
      si (r.a) < (s.b) alors r := tuple_suivant (R);
      sinon    si (r.a) > (s.b) alors s := tuple_suivant (S);
               sinon
                  sauv_s := s; /* sauvegarde du pointeur s dans le cas où l'attribut de
                                  jointure n'est pas une clé */
                  tant que (r.a = s.b et non_fin (S)) faire
                     écrire le tuple résultat dans tampon_res;
                     si tampon_res est plein alors écrire tampon_res sur disque dans T; fin si
                     s := tuple_suivant (S);
                  fin tant que
                  r := tuple_suivant (R); s := sauv_s;
               fin si
      fin si
   fin tant que
   écrire tampon_res sur disque dans T;
Fin
```

Figure II.13 : Jointure par tri-fusion

1. Une monotonie est définie comme un ensemble d'une ou plusieurs pages triées

La seconde phase (figure II.13) fusionne les deux relations en recherchant les tuples qui se joignent. Initialement, la phase de fusion charge le premier tuple de chaque relation. Lors de la comparaison des attributs de jointure a et b des tuples courants r et s, trois cas peuvent se présenter. Dans le cas où r.a est inférieur à s.b alors on passe au tuple suivant de R. En revanche, si r.a est supérieur à s.b alors on passe au tuple suivant de S. Enfin, si r.a est égale à s.b alors le tuple résultat est formé et on passe au tuple suivant de l'une des deux relations.

Plusieurs versions parallèles de l'algorithme de jointure par tri-fusion ont été proposées [VAL 84, SCH 89]. Une première version, décrite dans [VAL 84], consiste à effectuer un tri parallèle de R, puis un tri parallèle de S suivi d'une fusion des deux relations triées qui s'effectue sur un seul processeur. Nous décrivons donc uniquement la méthode de tri parallèle d'une relation R en supposant que R est déjà répartie sur les disques des d processeurs. L'algorithme parallèle de tri par fusion est constitué de deux étapes :

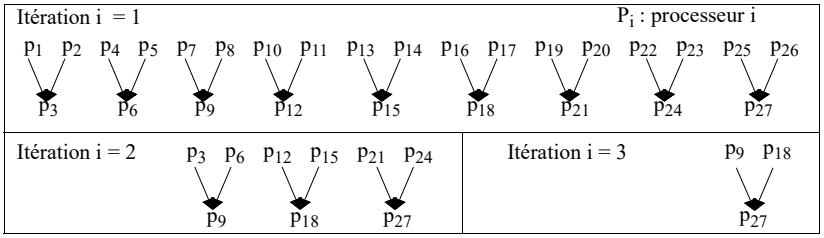
E1 : construction d'une seule monotonie par processeur. Sur chaque processeur est exécuté l'algorithme de tri par fusion développé précédemment pour obtenir une seule monotonie par processeur. Pour une présentation plus complète des algorithmes parallèles de tri parallèle nous recommandons aux lecteurs les références [AKL 85, COS 93, GIB 88] ;

E2 : fusion des d monotonies pour obtenir une relation triée sur un seul processeur (p_d) en suivant l'algorithme parallèle décrit ci-après. Pour faciliter la description de l'algorithme nous considérons que d est une puissance de b pour s'assurer qu'il y ait toujours (b-1) processeurs qui envoient leur monotonie sur 1 processeur.

```
Procédure de tri par fusion inter-processeur
Début
  pour i := 1 à log_b d faire
    pour chaque processeur P_{(k-1)b^i+j b^{i-1}} avec k ∈ [1, d/b^i] et j ∈ [1, b-1] faire en parallèle
      envoyer_monotonie (P_{(k-1)b^i+j b^{i-1}}, P_{kb^i});
    fin pour
    pour chaque processeur P_{kb^i} avec k ∈ [1, d/b^i] faire en parallèle fusionner_monotonies;
      /* chaque processeur P_{kb^i} fusionne les (b-1) monotonies reçues avec sa propre
       monotonie en une seule */
    fin pour
  fin pour
Fin
```

Figure II.14 : Tri par fusion inter-processeur

Le principe de l'algorithme de tri par fusion inter-processeur est illustré ci-dessous dans le cas où d = 27 processeurs et b = 3.



Une seconde version développée dans [SCH 89] consiste, dans une première étape, à *redistribuer* (si nécessaire) les relations R et S sur les disques de d processeurs en utilisant *la même fonction de hachage* appliquée sur les attributs de jointure. On obtient des sous-relations R_1, R_2, ..., R_d et S_1, S_2, ..., S_d. telles que $\forall i \in [1, d], \forall j \in [1, d]$ avec $i \neq j$ $R_i \cap R_j = \varnothing$ et $\forall i \in [1, d], \forall j \in [1, d]$ avec $i \neq j$ $S_i \cap S_j = \varnothing$. Dans une deuxième étape, sur chaque processeur P_i, l'algorithme de jointure par tri-fusion est appliqué : tri_fusion (R_i, S_i, T_i).

5. 3 Jointure par hachage

L'approche de jointure par hachage a été introduite pour réduire le nombre de comparaisons par rapport à l'approche de jointure par produit cartésien. Le principe d'une jointure par hachage est de construire d'abord une table de hachage avec la plus petite des deux relations, notée R. Ensuite, pour chaque tuple s de la relation S, on détermine l'entrée dans la table de hachage et on joint le tuple s avec tous les tuples appartenant à cette entrée. Dans cet esprit, plusieurs méthodes ont été proposées, analysées et expérimentées [AKL 87, BIT 83, DEW 93, DEW 84, DEW 85, QAD 88, KIT 83, LU 90, MEH 93, SCH 89, RIC 87]. Nous développons les trois principales méthodes : jointure par hachage simple, " Grace hash-join " et jointure par hachage hybride.

5. 3. 1 Jointure par hachage simple

L'algorithme de jointure par hachage simple est constitué de deux étapes :

E1 : construire " build " la table de hachage avec la relation R,

E2 : sonder " probe " avec la relation S la table de hachage.

Dans la première étape, la table de hachage est construite avec une fonction de hachage appliquée sur l'attribut de jointure. Les tuples de la relation R sont

ainsi répartis dans des paquets " bucket " composés d'une ou plusieurs pages. Une fonction de hachage h associe à une valeur de clé un entier représentant un numéro de paquet dans la table de hachage [DEL 82]. Tous les tuples dont la valeur de clé est telle que la fonction h associe un même numéro de paquet sont stockés ensemble dans ce paquet. Dans la seconde étape, la relation S est lue page par page et chaque tuple sonde la table de hachage afin de trouver les tuples pertinents pour calculer la jointure. Lors du sondage de la table de hachage, la fonction h appliquée au tuple s de la relation S sur l'attribut de jointure, détermine le paquet à joindre avec le tuple s. Ce dernier est comparé avec tous les tuples du paquet déterminé.

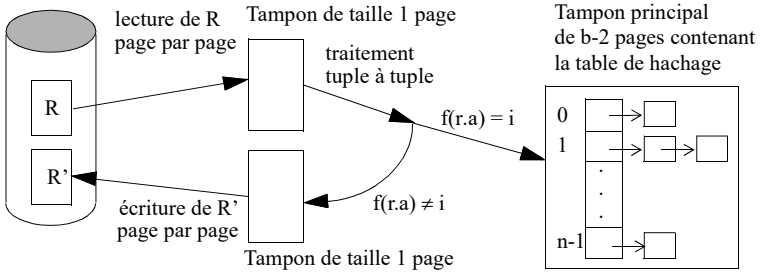


Figure II.15 : Construction de la table de hachage du $paquet_i$

Quand la table de hachage de la relation R ne peut pas tenir en mémoire (i.e. ||R|| > b-1 pages), alors la jointure doit s'effectuer par *morceaux*. Une fonction f[1] de hachage est utilisée pour déterminer les tuples appartenant au $paquet_i$ de la relation R à insérer dans la table de hachage de taille b-2 pages. Un $paquet_i$ est défini comme l'ensemble des tuples $r \in R$ vérifiant f(r.a) = i où r.a représente l'attribut de jointure. Pour chaque $paquet_i$, dans une première étape, tous les tuples de la relations R sont lus page par page et seuls les tuples du $paquet_i$ sont insérés dans la table de hachage avec une fonction g[2]. Quant aux autres tuples, ils sont écrits page par page sur disque (figure II.15). Dans une seconde étape, la relation S est balayée page par page et les tuples tels que f(s.b) = i sondent la table de hachage. Les autres tuples ne vérifiant pas f(s.b) = i sont écrits page par page sur disque (figure II.16). L'algorithme de jointure par hachage simple est donné dans la figure II.17.

1. Par exemple f(x) = x mod k où k représente le nombre de paquets. k est déterminé en fonction du nombre de pages de R et de la taille du tampon principal qui est égale à b-2 pages.
2. Les fonctions f et g doivent être différentes sinon la table de hachage associée à g n'aura qu'une seule entrée.

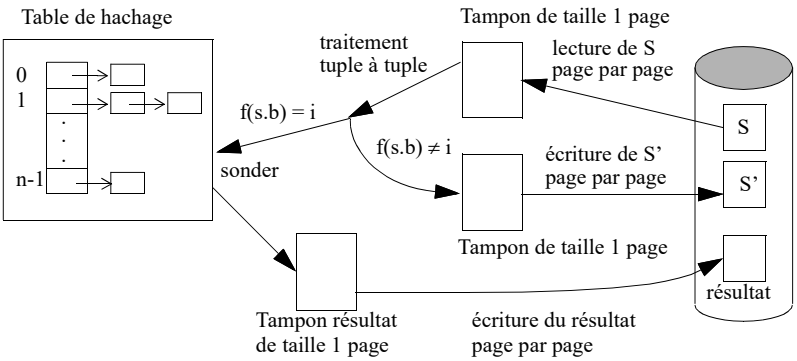


Figure II.16 : Sondage de la table de hachage du paquet$_i$

```
Procédure join_hachage_simple (R, S, T : relation)
Début
    pour chaque paquet_i faire
        pour chaque tuple r de R non traité faire
            si f(r.a) = i alors insérer (r, table_hachage);
            sinon
                écrire r dans tampon_R; /* taille (tampon_R) = 1 page */
                si le tampon_R est plein alors écrire tampon_R sur disque dans R'; fin si
            fin si
        fin pour
        écrire tampon_R sur disque dans R';
        pour chaque tuple s de S non traité faire
            si f(s.b) = i alors sonder (s, table_hachage, T);
            sinon
                écrire s dans un tampon_S; /* taille (tampon_S) = 1 page */
                si le tampon_S est plein alors écrire tampon_S sur disque dans S'; fin si
            fin si
        fin pour
        écrire tampon_S sur disque dans S';
        R := R';
        S := S';
        vider (table_hachage);
    fin pour
Fin
```

Figure II.17 : Jointure par hachage simple

Une variante de l'algorithme décrit ci-dessus consiste à éliminer la relation R' (suppression des écritures disque) et à balayer entièrement la relation R à chaque itération [LU 94].

5. 3. 2 " Grace hash-join "

L'inconvénient majeur de l'algorithme de jointure par hachage simple est que certains tuples de R ou de S peuvent être lus et écrits plusieurs fois. Pour résoudre ce problème, l'algorithme " Grace hash-join " a été introduit par [KIT 83] et dont le principe est décrit ci-après. La première étape répartie, en utilisant une fonction de hachage appliquée sur l'attribut de jointure, les deux relations R et S en une paire de N paquets R_1, R_2, ..., R_N et S_1, S_2, ..., S_N. Le nombre de paquets N est choisi pour être très grand. Ceci permet d'augmenter la probabilité que la mémoire soit suffisante pour effectuer la jointure de deux paquets. Dans le cas où les paquets sont plus petits que l'espace mémoire, alors plusieurs paquets peuvent *corésider* en mémoire. La deuxième étape, effectue la jointure de toutes les pairs de paquets R_i, S_i en appliquant l'algorithme de jointure par hachage simple sans contrainte mémoire.

```
Procédure Grace_hash_join (R, S, T : relation)
Début
    pour chaque tuple r de R faire écrire r dans le paquet R_f(r.a) sur disque; fin pour
    pour chaque tuple s de S faire écrire s dans le paquet S_f(s.b) sur disque; fin pour
    pour chaque paquet R_i faire
        pour chaque tuple r de R_i faire insérer (r, table_hachage);
        fin pour
        pour chaque tuple s de S_i faire sonder (s, table_hachage, T);
        fin pour
        vider (table_hachage);
    fin pour
Fin
```

Figure II.18 : Grace hash-join

L'algorithme " Grace hash-join " diffère fondamentalement de l'algorithme de jointure par hachage simple, par le fait que la construction des paquets R_i et S_i et la jointure des paquets sont complètement *dissociées*.

5. 3. 3 Jointure par hachage hybride

Dans l'algorithme Grace hash-join, les paquets R_1 et S_1 sont d'abord stockés sur disque, puis ils sont lus pour construire la table de hachage avec R_1 et la sonder avec S_1. Afin d'éviter ces écritures et lectures disque, l'algorithme de jointure par hachage hybride a été proposé par [DEW 84, SCH 89]. Le principe de cet algorithme (figure II.19) est que pendant le partitionnement de la relation en N paquets, la première table de hachage est construite avec les tuples appartenant au paquet R_1. De plus, pendant le partitionnement de la relation S les tuples appartenant au paquet S_1 sont utilisés pour sonder la table

de hachage. Dans cet algorithme le nombre de paquets est choisi afin que la taille de chaque paquet de R soit très proche (inférieurement) des b-2 pages du tampon principal. Les autres jointures des paquets R_i, pour i := 2 jusqu'à N, elles sont identiques à celles développées pour l'algorithme Grace hash-join.

```
Procédure jointure_hybride (R, S, T : relation)
Début
    pour chaque tuple r de R faire
        si f(r.a) = 1  alors insérer (r, table_hachage);
        sinon écrire r dans le paquet R_f(r.a) sur disque;
        fin si
    fin pour
    pour chaque tuple s de S faire
        si f(s.b) = 1  alors sonder (s, table_hachage, T);
        sinon écrire s dans le paquet S_f(s.b) sur disque;
        fin si
    fin pour
    pour chaque paquet R_i   faire /* avec i ∈ [2, N] */
        vider(table_hachage);
        pour chaque tuple r de R_i faire insérer (r, table_hachage); fin pour
        pour chaque tuple s de S_i faire sonder (s, table_hachage, T); fin pour
    fin pour
Fin
```

Figure II.19 : Jointure par hachage hybride

La différence entre l'algorithme Grace hash-join et l'algorithme de jointure par hachage hybride réside dans la construction des paquets. En effet, dans la jointure par hachage hybride la relation R est répartie afin qu'un paquet de R tienne exactement dans l'espace mémoire. Alors que dans le Grace hash join le nombre N de paquets est choisi pour être très important afin qu'un ou plusieurs paquets puissent tenir en mémoire. De plus, dans la jointure par hachage hybride, la construction de la première table de hachage s'effectue en même temps que le partitionnement de la relation R, et le partitionnement de S s'effectue aussi en même temps que le sondage de la première table. Ainsi, la jointure par hachage hybride évite d'une part, d'écrire les paquets R1 et S1 sur disque pendant la phase de partitionnement et d'autre part, de les relire pendant la phase de jointure.

5. 3. 4 Algorithmes parallèles de jointure par hachage

Les versions parallèles des algorithmes de jointure par hachage sont toutes fondées sur le même principe : (i) les relations R et S sont redistribuées, si nécessaire, sur les disques des processeurs participant à la jointure avec une

fonction de hachage appliquée sur les attributs de jointure, et (ii) sur chaque processeur est exécuté l'algorithme de jointure.

Dans un contexte mono-processeur, nous avons vu que les algorithmes de jointure par hachage nécessitent l'utilisation de deux fonctions de hachage : l'une pour découper une relation en plusieurs paquets, et l'autre pour construire la table de hachage. Pour paralléliser ces algorithmes il est nécessaire de répartir les relations opérandes sur les disques des processeurs, avec une nouvelle fonction de répartition de type hachage h. Il est évident que les trois fonctions f, g et h doivent être *deux à deux distinctes*. Cette condition permet d'éviter qu'on ait un seul paquet par processeur et qu'on ait une seule entrée dans la table de hachage.

5. 4 Quelques éléments de comparaison des algorithmes de jointure

Vu l'abondance et la richesse de la littérature en matière de proposition d'algorithmes séquentiels et parallèles pour implanter une jointure, nous nous sommes limités à la description d'une dizaine d'algorithmes séquentiels ainsi qu'à leurs versions parallèles. Si l'on est convaincu que le principal problème d'un SGBD n'est pas celui de la *faisabilité* mais celui des *performances* [BAN 86], alors les principales questions qui se posent sont : (i) comment choisir un ou plusieurs algorithmes de jointure à implanter dans un SGBD? et (ii) quels sont les éléments à intégrer dans un SGBD pour choisir à la compilation (ou à l'exécution) l'algorithme jointure le plus efficace en termes de minimisation du temps de réponse et d'utilisation optimale des ressources disponibles? Dans cette perspective plusieurs travaux ont été réalisés s'appuyant sur des modèles analytiques, des simulations et des expérimentations. Ces travaux ont permis d'analyser et de comparer des algorithmes séquentiels et des algorithmes parallèles [BIT 83, BOU 90, VAL 84, SCH 89, DEW 93, QAD 88, RIC 87]. Dans la suite, en s'appuyant sur les résultats de ces travaux, nous essayons de donner quelques caractéristiques permettant de comparer globalement les algorithmes de jointure décrits dans cette section.

L'avantage des algorithmes de jointure par produit cartésien est la *généralité*, dans le sens où tous les critères de jointure peuvent être traités. Cependant, le temps de réponse peut être prohibitif à cause, d'une part, du nombre très important de comparaisons puisque tous les tuples de R sont comparés à tous les tuples de S, et d'autre part, du nombre considérable d'Entrées/Sorties lorsque la plus petite des relations ne tient pas en mémoire. Ainsi ces algorithmes sont utilisés dans le cas où les relations à joindre sont de faible taille (par exemple quand la plus petite des relations tient en mémoire) ou s'il n'est pas possible de satisfaire le critère de jointure avec un autre algorithme.

Le temps de réponse des algorithmes de jointure par tri-fusion est fortement influencé par le *tri des deux relations*. Ces algorithmes deviennent les plus appropriés lorsqu'on souhaite récupérer le résultat trié sur l'attribut de jointure ou si l'une ou les deux relations sont déjà triées. De plus, ces algorithmes peuvent facilement être étendus à l'*inéqui-jointure* (utilisation des comparateurs <, <=, >, >=). Malheureusement, ils ne sont pas adaptables à l'opérateur de comparaison ≠.

L'existence d'un *index* sur l'une des deux relations à joindre rend, généralement, les algorithmes de jointure par produit cartésien indexé les plus performants. En revanche, si les deux relations sont initialement triées, cette supériorité peut être remise en cause. Quant à la généralité, ces algorithmes sont difficilement modifiables à faible coût pour considérer l'inéqui-jointure et l'opérateur ≠.

Les algorithmes de jointure par hachage s'avèrent souvent les plus performants si les relations ne sont pas triées, s'il n´existe pas d'index et si l'une des deux relations tient en mémoire. Le coût essentiel de ces algorithmes provient du hachage des données en paquets et de la construction des tables de hachage. Une des difficulté majeure de ces algorithmes est de trouver deux bonnes fonctions de hachage distinctes (i.e. f et g) qui assurent l'*équi-répartition* des données dans les paquets et dans chaque entrée de la table de hachage. En outre, ces algorithmes se limitent souvent à l'équi-jointure. En effet, ils sont difficilement adaptables à l'inéqui-jointure [BOU 90] puisque la relation d'ordre n'est généralement pas conservée lorsqu'on utilise une fonction de hachage. Par exemple, si l'on a $val_a<val_b$ alors cela n'implique pas toujours que $h(val_a)<h(val_b)$ [BOU 90].

A partir de ces observations relatives aux différents algorithmes de jointure, nous constatons qu'il n'existe pas d'algorithme supérieur indépendamment du profil de la base de données (taille des relations, index, ...) et de la taille mémoire. En conséquence, un SGBD doit posséder plusieurs algorithmes de jointure incluant l'algorithme de jointure par produit cartésien qui est le seul à s'appliquer à tous les critères de jointure. A charge de l'optimiseur de choisir le meilleur algorithme qui minimise une fonction de coût (par exemple le temps de réponse).

Dans un SGBD parallèle le problème de choix de l'algorithme de jointure le plus approprié est beaucoup plus complexe en raison de l'introduction d'une nouvelle dimension qui est le parallélisme. Cette dimension engendre de nouveaux et cruciaux problèmes, notamment :

1 - le choix d'une architecture parallèle. En effet, le type d'architecture parallèle influe sur la détermination du meilleur algorithme de jointure puisque la fonction temps de réponse d'un algorithme dépend de l'organisation des données en mémoire (mémoire commune, mémoire distribuée, ...) ;

2 - le choix d'une topologie de réseau d'interconnexion. La topologie du réseau a un impact direct sur les temps de communication de données et de contrôle ;

3 - les communications de données et de contrôle engendrées par l'exécution parallèle d'une requête. Le temps de communication peut devenir prohibitif, ce qui dégrade le temps de réponse et diminue l'apport du parallélisme ;

4 - le nombre de processeurs optimal alloués à une opération relationnelle. Il s'agit de déterminer le point de compromis traitement-communication. L'analyse des algorithmes de jointure montre que la courbe du temps de réponse en fonction du nombre de processeurs [CHE 92a] est constituée de deux parties. Dans la première partie de la courbe l'ajout de processeurs diminue rapidement le temps de réponse. Dans la seconde partie, au delà d'un certain nombre de processeurs, l'ajout d'un processeur augmente le temps de réponse. Ceci est dû au fait que le temps de communication est supérieur au temps de traitement gagné ;

5 - l'équilibrage de charge sur tous les processeurs. La difficulté est de trouver une fonction de répartition garantissant l'équi-répartition des données pour assurer une rentabilité maximale des capacités du système parallèle.

6 Conclusion

Dans ce chapitre, par le biais de diverses applications (banque, assurance, hôtellerie, ...) nous avons voulu montrer la nécessité d'exploiter le parallélisme et les récentes architectures parallèles pour obtenir de hautes performances [DEW 92, VAL 93]. Celles-ci consistent à assurer le meilleur rapport coût/performance, à améliorer le temps de réponse des requêtes et à augmenter les capacités du système. Le résultat le plus important obtenu est l'ordre d'influence des formes de parallélisme sur l'optimisation du temps de réponse (temps de traitement+ temps des E/S + temps des communications) et sur l'amélioration des capacités du système parallèle.

Trois familles de SGBD parallèles ont été conçues, prototypées et réalisées : les SGBD à mémoire commune, les SGBD à disques partagés, et les SGBD à mémoire distribuée. Les trois architectures parallèles, utilisées comme plateforme pour implanter un SGBD parallèle, présentent des avantages et des

inconvénients par rapport à ensemble de paramètres qui sont : l'extensibilité, le coût des communications, la bande passante en entrées/sorties, la fiabilité et la disponibilité. Une question de fond qui se pose, pour implanter un SGBD parallèle, est le choix de l'architecture la plus appropriée en tenant compte de l'évolution des données et de la nature des applications qui peuvent être décisionnelle ou transactionnelle.

La parallélisation des algorithmes des opérations relationnelles requiert la répartition des relations de base en sous-relations. Pour cela, deux approches ont été proposées : le partitionnement total et le partitionnement partiel. Cette dernière est plus difficile à réaliser et administrer à cause de la complexité pour estimer le degré de répartition des relations de base ou temporaire afin d'assurer le meilleur compromis traitement-communication et l'utilisation efficace des ressources du système parallèle.

Les différentes approches et méthodes de répartition des relations permettent de paralléliser les opérateurs relationnels et en particulier l'opérateur de jointure dont les performances sont déterminants pour la viabilité de tout SGBD. Trois classes d'algorithmes parallèles, fondées sur le produit cartésien, le tri-fusion et le hachage ont été proposées, implantées, analysées et comparées. Au delà de la comparaison des algorithmes de jointure au travers des modèles analytiques et des expérimentations pour montrer la supériorité de certains algorithmes sur d'autres, un consensus général se dégage pour concevoir et réaliser un *évaluateur de coûts* [AND 91, GAN 92, HAM 92c, HAM 95a, ZIA 92] dont le rôle, entre autres, est de choisir l'algorithme le plus approprié à l'opération relationnelle considérée.

7 Références bibliographiques

[ABD 95] A. Abdellatif et al., " Oracle7 langages - architecture - administration ", Eyrolles, 1995.

[AHO 89] A. Aho et al., " Structures de données et algorithmes ", Ed. InterEditions, 1989.

[AKL 85] S. Akl, " Parallel Sorting Algorithms ", Academic Press, New York, 1985.

[AKL 87] S. G. Akl, N. Santroo, " Optimal Parallel Merging and Sorting Without Memory Conflicts ", IEEE Trans. and Computers, Vol. 36, No. 11, 1987, pp. 1367-1369.

[AND 91] F. Andrès et al., " A Multi-Environment Cost Evaluator for Parallel Database Systems ", 2nd Intl. Symp. on Database Systems for Advanced Applications, Tokyo, April 1991.

[APE 92] P. Apers et al. " PRISMA/DB: A Parallel Main-Memory Relational DBMS ", IEEE, Trans. Knowledge and Data Engineering, Vol. 4, No. 6, Dec. 1992, pp. 541-554.

[BAN 86] F. Bancilhon, R. Ramakrishnan, " An Amateur's Introduction to Recursive Query Processing Strategies ", Proc. ACM SIGMOD Conf. on Management of Data, Washington, May, 1986.

[BAR 95] C. Baru et al., " An Overview of DB2 Parallel Edition ", Proc. ACM SIGMOD Conf. on Management of Data, San Jose, May 1995, CA, pp. 460-462.

[BER 91] B. Bergsten et al., " Prototyping DBS3: a Shared-Memory Parallel Database System ", First Intl. Conf. on Parallel Distributed Information Systems, Dec. 1991 Florida, USA, pp. 226-234.

[BIT 83] D. Bitton et al., " Benchmarking Database Systems - a Systematic Approach ", Proc. of the 1983 VLDB Conf., Oct. 1983, pp. 8-19.

[BOR 90] H. Boral et al., " Prototyping Bubba, a Highly Parallel Database System ", IEEE Trans. Knowledge and Data Engineering. Vol. 1, No.1, March 1990, pp. 4-24.

[BOU 90] M. Bouzeghoub et al., " Systèmes de bases de données: des techniques d'implantation à la conception de schémas ", Ed. Eyrolles, 1990.

[BRA 84] K. Bratbergsengen, " Hashing Methods and Relational Algebra Operations ", 10th VLDB Conf., Singapore, Août, 1984.

[CAR 92] F. Cariño, P. Kostamaa, " Exegesis of DBC/1012 and P-90 - Industrial Supercomputer Database Machines ", 4th Intl. PARLE Conf., Paris, June 1992, pp. 877-892.

[CHA 92] C. Chachaty et al., " A Compositional Approach for the Design of a Parallel Query Processing Language ", 4th Intl. PARLE Conf., Paris, June 1992, pp. 825-840.

[CHE 92a] M.S. Chen et al., " Scheduling and Processor Allocation for Parallel Execution of Multi-join Queries ", Proc. 8th Intl. Conf. Data Eng., Arizona, 92, pp. 58-67.

[COP 88] G. Copeland et al., " Data Placement in Bubba ", Proc. ACM SIGMOD Conf. on Management of Data, Chicago, May 1988, pp. 99-108.

[COS 93] M. Cosnard, D. Trystram, " Algorithmes et architectures parallèles ", Ed. InterEditions, Paris, 1993.

[DEL 82] C. Delobel, M. Adiba, " Bases de données et systèmes relationnels ", Ed. Dunod, Informatique, 1982.

[DEW 84] D. Dewitt et al., " Implementation Techniques for Main Memory Database Systems ", Proc. ACM SIGMOD Conf. on Management of Data, Boston, June 1984, pp. 1-8.

[DEW 85] D. Dewitt, R. Geber, " Multiprocessor Hash-based Join Algorithms ", Proc. of the 11th Intl. Conf. on VLDB, Stockholm, August 1985, pp. 151-162.

[DEW 90] D.J. Dewitt et al., " The Gamma Database Machine Project ", IEEE Trans. Knowledge and Data Engineering. Vol. 2, No. 1, March 1990, pp. 44-61.

[DEW 92] D.J. Dewitt, J. Gray, " Parallel Database Systems: The future of High Performance Database Systems ", Com. of the ACM, Vol. 35, No. 6, June 1992, pp. 85-98.

[DEW 93] D.J. Dewitt et al., " Nested loop Revisited ", Proc. Second Parallel and Distributed Information Systems Conf., IEEE CS Press, Los Alamitos, California, 1993, pp. 230-242.

[ENG 95] S. Englert et al., " Parallelism and its Price : A Case Study of NonStop SQL/MP ' SIGMOD Record, Vol. 24, No. 4, Dec. 1995.

[FLE 95] F. T. Fleighton, " Introduction aux algorithmes et architectures parallèles ", Traduction de P. Fraigniaud et E. Fleury, Morgan and Kaufmann Publisher, Inc. 1995.

[FLY 72] M.J. Flynn, " Some Computer Organisation and their Effectivenesses ", IEEE Trans. Computers, C-21, pp. 948-960, September 1972.

[GAN 92] S. Ganguly et al., " Query Optimization for Parallel Execution ", Proc. ACM SIGMOD Conf. on Management of Data, New York, june 1992, pp. 9-18.

[GER 91] C. Germain-Renaud, J.P. Sansonnet, " Les ordinateurs massivement parallèles " Ed. Armand Colin, Paris, 1991.

[GHA 90] S. Ghandeharizadeh, D.J. Dewitt, " Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines ", Proc. of the 16th Intl. Conf. onVLDB, Brisbane, 1990, pp. 481-492.

[GIB 88] A. Gibbons, W. Rytter, " Efficient Parallel Algorithms ", Cambridge University Press, 1988.

[GRA 90] G. Graefe, " Encapsulation of Parallelism in the Volcano Query Processing System ", Proc. ACM SIGMOD Conf. on Management of Data, Atlantic City, May 1990, pp. 102-111.

[INF 95] Informix Software, Inc., " Informix-OnLine Dynamic Server ", Version 7.1, Technical Brief, Menlco Park, California, 1995.

[HAM 92b] A. Hameurlain et al., " An Analytical Method to Allocate Processors in High Performance Parallel Execution of Recursive Queries ", Intl. Conf. on Database and Expert Systems Applications, Spring Verlag, Valencia, Sept. 1992, pp. 44-47.

[HAM 92c] A. Hameurlain et al., " A Cost Evaluator for Parallel Deductive Database Systems ", Intl. Conf. and Exhibition on Parallel Computing and Transputer Applications, IOS Press, Barcelona, Sept. 1992, pp. 1222-1228.

[HAM 95a] A. Hameurlain, F. Morvan, " A Cost Evaluator for Parallel Database Systems ", 6th Intl. Conf. on Database and Expert Systems Applications, LNCS 978, London, Sept. 1995, pp. 146-156.

[HIL 85] W. D. Hillis, " The Connection Machine ", Cambridge, MA, The MIT Press, 1985.

[HON 92] W. Hong, " Exploiting Inter-Operation Parallelism in XPRS ", Proc. ACM SIGMOD Conf. on Managment of Data, New York, June 1992, pp. 19-28.

[HUA 91] K. A. Hua et al., " Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing ", First Intl. Conf. on Parallel Distributed Information Systems, Florida, Dec. 1991, pp. 262-270.

[JAC 93] J.L. Jacquemin, " Informatique parallèle et systèmes multiprocesseurs ", Ed. Hermes, Paris, 1993.

[KIT 83] M. Kitsuregawa et al., " Application of Hash to Data Base Machine and its architecture ", New Generation Computing, Vol. 1, No. 1, 1983, pp. 63-74.

[KNU 73] D. Kunth, " The Art of Computer Programming: Sorting and Searching ", Addition Wesley Publishing Company, 1973.

[KOR 88] H. F. Korth, A. Silberschatz, " Systèmes de gestion des bases de données ",Traduction B. Loubières, Ed. MacGRAW-HILL, Paris, 1988.

[LIV 87] M. Livny et al., " Multi-Disk Management ", ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems, Banff Alberta, May 1987, pp. 69-77.

[LU 90] H. Lu et al., " Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory ", Proc. 16th Intl. Conf. VLDB, California, 1990, pp. 198-209.

[LU 94] H. Lu et al. " Query Processing in Parallel Relational Database Systems ", IEEE Computer Society Press, ISBN 0-8186-5452-X, Los Alamitos, CA, 1994.

[MEH 93] M. Mehta et al., " Batch Scheduling in Parallel Database Systems ", 9 th Conf. on Data Engineering, Vienna, Austria, April 1993, pp. 400-410.

[NIC 91] J. C. Nicolas, " Machines bases de données parallèles : contribution aux problèmes de la fragmentation et de la distribution ", Doct. Info., Lille, Jan. 1991.

[QAD 88] G. Z. Qadah, K. B. Irani, " The Join Algorithms on a Shared-Memory Multiprocessor Database Machine ", IEEE Trans. Software Engineering, Vol. 14, No. 11, Nov. 1998, pp. 199-214.

[RIC 87] J. P. Richardson et al., " Design and Evaluation of Parallel Pipelined Join Algorithms ", Proc. ACM SIGMOD Conf. on Management of Data, New York, 1987, pp. 399-409.

[SCH 89] D. Schneider, D. Dewitt, " A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment ", Proc. ACM SIGMOD Conf. on Management of Data, Portland, June 1989, pp. 110-121.

[SCH 90] D. Schneider, D. Dewitt, " Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines ", Proc. of the 16th VLDB Conf., Brisbane, Australia, 1990, pp. 469-480.

[STO 88] M. Stonebraker et. al., " The Design of XPRS ", Proc. of the14 th VLDB Conf., San Mateo, Calif., 1988, pp. 318-330.

[ULL 85] J. D. Ullman, " Implementation of Logical Query Languages for Databases ", ACM Trans. Database Systems, Vol. 10, No. 3, 1985, p. 289-321.

[VAL 84] P. Valduriez, G. Gardarin, " Join and Semi-join Algorithms for a Multiprocessor Database Machine ", ACM TODS, Vol. 9, No. 1, March 1984, pp. 133-161.

[VAL 93] P. Valduriez, " Parallel Database Systems: Open Problems and News Issues ", Distributed and parallel Databases, Kluwer Academic Publishers, Vol. 1, No. 2, 1993, pp. 137-165.

[WIT 93] A. Witkowski, "NCR 3700 the Next-Generation Industrial Database Computer", Proc. of the 19th VLDB Conf., Dublin, Ireland, 1993, pp. 230-243.

[ZIA 92] M. Ziane, " Optimisation de requêtes pour un système de gestion de bases de données parallèle ", Doctorat de l'Université de PARIS 6, Février 1992.